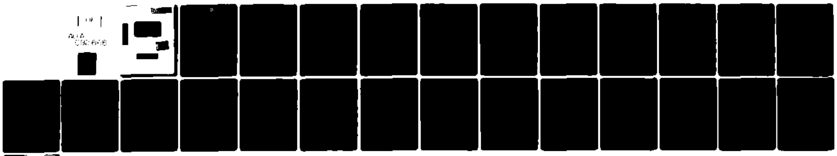
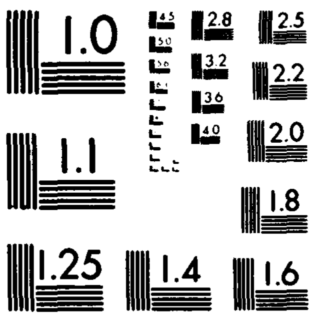



1.0
4.5
5.0
5.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

BOOLEAN VARIABLES IN REGULAR EXPRESSIONS AND FINITE AUTOMATA*

Karl R. Abrahamson
Department of Computer Science
University of Washington
Seattle, WA 98195

Technical Report 80-08-02 August 1980

*This research was supported in part by the National Science Foundation under Grant MCS77-02474 and by the Office of Naval Research under Contract N00014-80-C-0221.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR ~~80-08-02~~ ✓	2. GOVT ACCESSION NO. AD-A092 666	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) BOOLEAN VARIABLES IN REGULAR EXPRESSIONS AND FINITE AUTOMATA		5. TYPE OF REPORT & PERIOD COVERED 9 Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Karl R. Abrahamson		8. CONTRACT OR GRANT NUMBER(s) ONR Contract: N00014-80-C-0221 [redacted] NSF ~~Grant~~ MCS77-02474
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science, FR-35 University of Washington Seattle, WA 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 049-456/30 Oct 79 (437)
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research 800 North Quincy Street Arlington, VA 22217	National Science Found. Math. & Comp. Sciences Washington, D.C. 20550	12. REPORT DATE August 1980
		13. NUMBER OF PAGES 25
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Dr. R. B. Grafton		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Regular sets, succinctness, Boolean variables.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper considers questions of succinctness of representation of regular sets by regular expressions and finite automata which may contain special instructions for setting and testing auxiliary Boolean variables.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

1. INTRODUCTION

Although every regular set is described by some regular expression, there may not be any reasonably short regular expression for a given regular set. Therefore, people tend to use more powerful means of describing sets. For example, the complementation and intersection operators can be used to form concise representations of some sets (see Stockmeyer (1974)).

Auxiliary variables are commonly used informally in regular expressions (RE), and especially finite automata (FA), to help shorten the representation of a set. For example, consider a lexical scanner of a compiler, which is commonly implemented as a finite state machine. Suppose a certain language permits decimal integers of the form Dn , and binary integers of the form Bn . An ordinary finite state machine recognizing either decimal or binary integers would read either D or B , and then fork into two separate machines, one accepting decimal digits, the other accepting binary digits. But by adding an auxiliary Boolean variable, call it dec, the binary and decimal integer readers can share the same states. The machine described below recognizes decimal and binary integers.

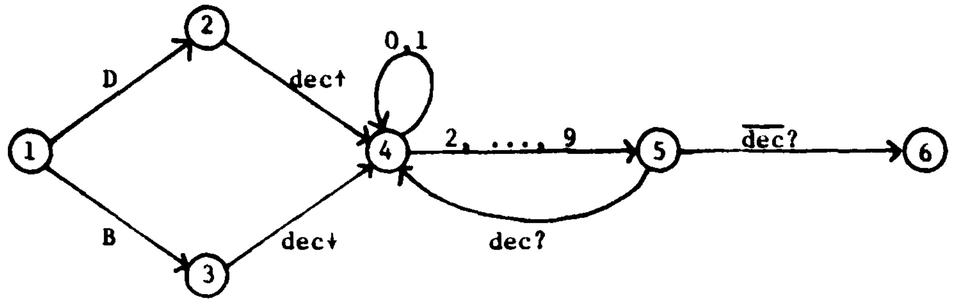
```
State 1:  on input B, set dec := false, goto 2;
          on input D, set dec := true, goto 2.
State 2:  on input 0 or 1, goto 2;
          on input 2,...,9, if dec then goto 2,
                            else stop and reject;
          on end of string, stop and accept;
          on any other symbol, stop and reject.
```

The example above is very simple, and the reader may consider the ordinary finite state machine, without auxiliary variables, more appealing. However, the reader can no doubt supply examples from his own experience in which a few well chosen variables greatly reduce the length of a program, even one which is essentially finite state.

In this paper we formally define regular expressions and finite automata with auxiliary Boolean variables, and prove some results concerning conciseness of representation of sets in the various models (ordinary RE, ordinary FA, RE with auxiliary Boolean variables and FA with auxiliary Boolean variables).

We begin with an informal description of RE and FA with auxiliary Boolean variables (BRE and BFA, respectively). There are two basic types of operations on a variable: setting it to a particular value, and testing whether it contains a particular value. For Boolean variables, there are just four operations: $x\uparrow$ (set x to _true_), $x\downarrow$ (set x to _false_), $x?$ (is x _true_?), and $\bar{x}?$ (is x _false_?). Each may appear in a BFA or BRE wherever an alphabetic symbol may normally appear. (Although $x\uparrow$, $x\downarrow$, $x?$ and $\bar{x}?$ are written as strings of two or more characters, we think of them as indivisible symbols.) In a BFA, a transition labeled $x\uparrow$ ($x\downarrow$) acts as a null move, with the side effect of setting the value of x to _true_ (_false_). A transition labeled $x?$ ($\bar{x}?$) may be taken, as a null move, provided x is currently _true_ (_false_). When x is _false_ (_true_), $x?$ ($\bar{x}?$) transitions may not be taken.

The following BFA recognizes the decimal and binary integers described above.



State 1 is the start state, and State 4 is the accepting state.

The special forms $x\uparrow$, $x\downarrow$, $x?$ and $\bar{x}?$ can be added to RE as well as FA. One way to understand BRE is to imagine a BFA "executing" the BRE. In that view, when $x\uparrow$ ($x\downarrow$) is encountered, no symbol is generated, but x is set <u>true</u> (<u>false</u>). When x is <u>true</u> (<u>false</u>), $x?$ ($\bar{x}?$) is treated as λ, the null string. When x is <u>false</u> (<u>true</u>), $x?$ ($\bar{x}?$) may not be generated; the generation must be aborted.

Alternatively, $x\uparrow$, $x\downarrow$, $x?$ and $\bar{x}?$ can be generated just as if they were alphabetic symbols. The strings generated in such a manner by a BRE must be passed through a filter, which filters out any string which contains $x\uparrow\ t\ \bar{x}?$ or $x\downarrow\ t\ x?$ as a substring, where t does not contain $x\uparrow$ or $x\downarrow$. As all variables are arbitrarily assumed to be <u>false</u> initially, strings containing substrings of the form $tx?$ must also be filtered out. After filtration, all occurrences of $x\uparrow$, $x\downarrow$, $x?$ and $\bar{x}?$ are erased. The formal definition of BRE given here uses this filtration approach.

Boolean variables can be used to simulate some useful concepts.

1. Integers in the range $[0, 2^n - 1]$ can be stored in n Boolean variables. It is a straightforward exercise to write "programs" which increment variables mod 2^n, or add them mod 2^n, or test for zero. A "program" in this context is a BRE or BFA, depending on which model is being

used, whose language is $\{\lambda\}$, and which has a desired side effect on some Boolean variables. For example, suppose x is stored in $x_n\, x_{n-1} \ldots x_1$. "$x \leftarrow x + 1 \bmod 2^n$" is accomplished by the program

$$y\uparrow \cdot (\bar{y}? \cup y?(x_1? \cdot x_1\downarrow \cup \bar{x}_1? \cdot x_1\uparrow \cdot y\downarrow))$$

$$\cdot \quad \ldots$$

$$\cdot\ (\bar{y}? \cup y?(x_n?\ x_n\downarrow \cup \bar{x}_n?\ x_n\uparrow\ y\downarrow))\ .$$

The extra variable y holds the carry bit. Each line does one stage of a ripple through increment.

2. It is possible to write a BRE for $E^n = E \cdot \ldots \cdot E$ (n times), which has length only $\ell(E) + O(\log n)$. Using $\lfloor \log n \rfloor + 1$ variables to hold an integer x in the range $[0, n]$, we write E^n as

$$(x \leftarrow 0) \cdot ((x \neq n?) \cdot E \cdot (x \leftarrow x + 1))^* \cdot (x = n?)\ .$$

3. Subroutine calls to bounded depth with bounded parameters can be implemented in BFA. Simply store a "return address" in certain variables before transferring to a certain section of the BFA. When the return state is encountered, test the value of the return address. In Section 3 we show that BRE can efficiently (i.e. with a factor of $\log n$ size increase) simulate BFA. Using that simulation, subroutines can be implemented in BRE.

Programs in Propositional Dynamic Logic (PDL) are very close to regular expressions. $\cdot$ is the sequencing operator, $\cup$ the nondeterministic choice operator, and * the looping operator. Boolean variables can be added to such programs very much the way they are added to regular expressions. The programs $x\uparrow$ and $x\downarrow$ are thought of as assignment programs. The extension B-PDL (Boolean PDL) of PDL is studied in Abrahamson (1980). There Boolean variables are used to express properties in B-PDL which are also expressible in PDL, but not obviously so.

2. FORMAL DEFINITIONS OF BOOLEAN REGULAR EXPRESSIONS AND FINITE AUTOMATA

Let B be a finite set of Boolean variables, and let Σ be a finite alphabet. Let $\Gamma = \{x\uparrow, x\downarrow, x?, \bar{x}? : x \in B\}$. A Boolean regular expression (BRE) over alphabet Σ and variables B is any well formed expression formed from members of $\Sigma \cup \Gamma$, the dyadic operators $\cup$ and $\cdot$, and the monadic operator *. The unfiltered language $U(E)$ of expression E is defined inductively by the following rules.

$U(\sigma) = \{\sigma\}$ for $\sigma \in \Sigma \cup \Gamma$,

$U(E \cup F) = U(E) \cup U(F)$,

$U(E \cdot F) = U(E) \cdot U(F)$ (concatenation of sets),

$U(E^*) = U(E)^*$ (Kleene closure).

Let $\Delta = \Sigma \cup \Gamma$ and $\Delta_x = \Delta - \{x\uparrow, x\downarrow\}$. R_x is the set of strings which are inconsistent with respect to variable x, and is defined by the regular expression

$$R_x = (\Delta^* \cdot x\uparrow \cdot \Delta_x^* \cdot \bar{x}? \cdot \Delta^*) \cup (\Delta^* \cdot x\downarrow \cdot \Delta_x^* \cdot x? \cdot \Delta^*) \cup (\Delta_x^* \cdot x? \cdot \Delta^*).$$

The language $L(E)$ of BRE E is defined by

$$L(E) = H(U(E) - \bigcup_{x \in B} R_x),$$

where H just erases members of Γ from strings.

Example. The expression $x\uparrow \cdot y\uparrow \cdot (y? \cdot E \cdot (x? \cdot x\downarrow \cup \bar{x}? \cdot y\downarrow))^* \cdot \bar{y}?$, where E is any expression, represents the same language as $E \cdot E$. Note that E appears only once. If E is a large expression, the expression given here is nearly half as short as $E \cdot E$. The reader may find it helpful to generate $E \cdot E$ from this expression.

A Boolean finite automaton (BFA) is a 6-tuple $(V, \Sigma, B, \delta, Q, v_0)$, where

V is a set of vertices, or states,

$v_0 \in V$ is a distinguished start state,

$Q \subseteq V$ is a set of accepting states,

Σ is a finite alphabet,

B is a finite set of Boolean variables,

$\delta \in V \times (\Sigma \cup \Gamma) \times V$ is a set of transitions, or arcs.

A computation of machine $M = (V, \Sigma, B, \delta, Q, v_0)$ is a finite sequence C of members of δ beginning on (v_0, σ, u) for some $\sigma \in \Sigma \cup \Gamma$ and $u \in V$, ending on (u, σ, v) for some $u \in V$, $\sigma \in \Sigma \cup \Gamma$ and $v \in F$, and satisfying the constraints

1) if $(u, x\uparrow, u')\ t\ (v, \bar{x}?, v')$ is a contiguous subsequence of C, then t contains $(w, x\downarrow, w')$ for some w and w';

2) if $(u, x\downarrow, u')\ t\ (v, x?, v')$ is a contiguous subsequence of C, then t contains $(w, x\uparrow, w')$ for some w and w';

3) if $t(v, x?, v')$ is a prefix of C, then t contains $(w, x\uparrow, w')$ for some w and w'.

4) if $(u, \sigma, v)\ (w, \tau, x)$ is a subsequence of C, then $v = w$.

The string accepted by C is the sequence of all members of Σ which appear in C, in the order in which they appear. The language $L(M)$ accepted by M is

$L(M) = \{s \in \Sigma^* : s$ is accepted by some computation of $M\}$. Machine M is deterministic if for every state u,

1) if $(u, \sigma, v) \in \delta$ for $\sigma \in \Sigma$, then δ does not contain (u, σ, v') or (u, τ, v'') for any $v' \neq v$, $\tau \in \Gamma$ and any v''.

2) if $(u, \tau, v) \in \delta$ for $\tau \in \Gamma$, then (u, τ, v) is the only member of δ with first state u. A deterministic BFA is called a DBFA, while a nondeterministic BFA is called a NBFA.

3. RESULTS

Here is a summary of the results contained in this section. Two machines or expressions, or a machine and an expression, are equivalent if they represent (or recognize) the same language.

1. (Relations between BRE, DBFA and NBFA) We find a remarkably close relationship between BRE, DBFA and NBFA, much closer than that between RE, DFA and NFA. Recall that an exponential blowup results in general in converting a NFA to a RE, (see Ehrenfeucht and Zeiger (1976)), or in converting a NFA to a DFA (see Aho, Hopcroft and Ullman (1974)).

   a) For every length n BRE with m Boolean variables, there is an equivalent $O(n)$ state BFA with m Boolean variables.

   b) For every n state, b arc NBFA with m Boolean variables, there is an equivalent length $O(b \log n)$ BRE with $m + O(\log n)$ Boolean variables.

   c) For every n state, b arc NBFA with m Boolean variables, there is an equivalent $O(b)$ state DBFA with $O(m + n)$ Boolean variables.

2. (Relations between BFA and FA, and between BRE and RE) We find that Boolean variables permit concise representations of some languages. Let N be the maximum over all BFA's with n states and m distinct Boolean variables of the number of states in a minimal equivalent FA. Similarly, let K be the

maximum over all BRE's with length n and m distinct Boolean variables of the length of the minimal equivalent RE.

a) $N \leq n2^m$,

b) $N \geq c^n$ for some $c > 1$ and some $m = O(n)$

c) $K \leq c^{n2^m}$ for some c ,

d) $K \geq 2^{2^{dn}}$ for some $d > 0$, for some $m = O(n)$.

Theorems are numbered corresponding to the above list.

Theorem 1a. For every length n BRE E containing m Boolean variables, there is an $O(n)$ state BFA F which also contains m Boolean variables, such that $L(F) = L(E)$.

Proof. The proof is almost identical to the classical proof of the same theorem for $m = 0$, which can be found in Aho, Hopcroft and Ullman (1974).

Theorem 1b. For every NBFA F with n states, b transitions and m distinct Boolean variables there is a length $O(b \log n)$ BRE E with $m + O(\log n)$ Boolean variables, such that $L(E) = L(F)$.

Proof. Let the states of F be numbered $0, \ldots, n-1$. Let $x_1, \ldots, x_k$, $k = \lceil \log n \rceil$, be Boolean variables not appearing in F , which are used to represent an integer x in the range $[0, \ldots, n-1]$. E has the form $S \cdot \left(\bigcup_{i=1}^{b} T_i \right) \cdot R$. S sets x to the number of v_o . Let (u, σ, v) be the i^{th} transition of F . T_i is a sequence of three actions: 1) test if $x = u$, 2) generate σ , 3) set $x = v$. Finally R tests if $x \in Q$. We may assume without loss of generality that $b \geq n$, for otherwise some states are inaccessible. E has length $O(b \log n)$, and has $m + O(\log n)$ Boolean variables. It is not difficult to see that $L(E) = L(F)$.

Corollary 1. Every regular set is representable by a BRE which contains only one * symbol.

That contrasts with ordinary RE's, which require an unbounded number of stars for full representational power (see Cohen (1970), or Ehrenfeucht and Zeiger (1976)).

Proof. The expression in the proof of Theorem 1b has only one * symbol.

Theorem 1c. For every NBFA F with n states, b arcs and m distinct Boolean variables, there is a DBFA F' equivalent to F , with $O(b)$ states and $O(m + n)$ Boolean variables.

Proof. F' performs a deterministic simulation of F . F' contains a Boolean variable s_u for every state u in F . s_u is true at a given point in the simulation if F could be in state u .

F' begins by setting s_{v_o} = true , and s_v = false for all $v \neq v_o$. Then F' enters a loop in which it repeatedly performs Step 1, then Step 2, until the input is exhausted.

Step 1. Perform Boolean variable operations. Before reading the next character, F' needs to simulate F along all paths on which no characters are read. There is a problem here: F may contain a loop containing no read operations, only Boolean variable operations. To avoid following such loops forever, F' maintains a count r in variables $r_1, \ldots, r_k$, $k = \lceil \log(n2^m) \rceil$, of the number of consecutive Boolean variable operations of F which it performs.

When r reaches $n2^m$, F' ignores all further Boolean variable operations, and terminates Step 1. It should be clear to the reader that aborting non-reading loops after $n2^m$ steps cannot change the language of F. After all, if F follows a length $n2^m$ path, then some state must have been reached twice in the path with exactly the same Boolean variable values both times. So there is a shorter path which reaches the same state, with the same Boolean variable values, as the longer path.

Step 1 is defined by the following program. Let δ be the transition function of F.

```
for $r := 0$ to $n2^m - 1$ do
   for every state $u$ of $F$ do
      if $s_u$ then begin
         $s_u\downarrow$ ;
         for every $(u, \tau, v) \in \delta$, where $\tau$ is $x\uparrow$ or $x\downarrow$ for
                                                             some $x$, do
            begin $\tau$ ; $s_v\uparrow$ end;
         for every $(u, x?, v) \in \delta$ do
            if $x$ then $s_v\uparrow$ ;
         for every $(u, \bar{x}?, v) \in \delta$ do
            if not $x$ then $s_v\uparrow$ ;
         if $(u, \sigma, v) \in \delta$ for some $\sigma \in \Sigma$, then $s_u\uparrow$
      end
   endfor
endfor.
```

The reader should be able to convince himself that the above program correctly simulates the Boolean variable operations of F.

Step 2. Perform a read operation. Step 2 is defined by the following program, using auxiliary variables s'_u for every state u of F.

```
For every $\sigma \in \Sigma$ do
   if the next input character is $\sigma$ then
      for every state $u$ of $F$ do
         if $s_u$ then
            for every $(u, \sigma, v) \in \delta$ do $s'_v \uparrow$ ;
for every state $u$ in $F$ do $s_u \leftarrow s'_u$ .
```

We may assume without loss of generality that $b \geq n$ and $b \geq m$. Then machine F described above has $O(b)$ states and $m + \lceil \log n2^m \rceil + 2n + c$ Boolean variables (c variables might be used to implement the for loop on σ).

The simulation used in the proof of Theorem 1c is based on a commonly used simulation of NFA, in which the simulating machine keeps a record of which states the NFA can be in (see Aho, Hopcroft and Ullman (1974)). Theorem 1c presents another view of that simulation: convert the NFA to a DBFA, and run the DBFA.

Theorem 2a. Every n state NBFA F with m Boolean variables is equivalent to some $n2^m$ state FA F'.

Informal Proof. F' can be constructed as follows. First, make 2^m copies of F, one for each subset of the Boolean variables in F. Eliminate $x\uparrow$ arcs by replacing arc $(u, x\uparrow, v)$ in copy $s \subseteq B$ by a λ-arc from u in copy s to v in copy $s \cup \{x\}$. Similarly, replace $(u, x\downarrow, v)$ by a λ-arc from u in copy s to v in copy $s - \{x\}$. Replace $(u, x?, v)$

in copy s by a λ-arc , still within copy s , iff $x \in s$. If $x \notin s$, delete $(u, x?, v)$ from copy s . $\bar{x}?$ arcs are eliminated similarly. When all arcs not labeled by members of Σ have been eliminated, eliminate λ-arcs by the standard method (no states need be added). It is left to the reader to verify that machine F' so constructed recognizes the same language as F .

Theorem 2b. For all sufficiently large n there is an n state DBFA F which is not equivalent to any NFA F' of fewer than c^n states for some c .

Proof. Let F_k be the machine which implements the following program using k Boolean variables to represent an integer in the range $[0, 2^k - 1]$:

```
for i := 0 to $2^k - 1$ do (read a ).
```

F_k recognizes a^{2^k} , and has length dk for some d . Clearly F_k is not equivalent to any NFA of fewer than 2^k states. Let $F = F_{\lfloor \frac{n}{d} \rfloor}$, padded to exactly n states. Then F is equivalent to no NFA with fewer than c^n states for $c = 2^{\frac{1}{d} + \varepsilon}$.

Theorem 2c. Every length n BRE E with m Boolean variables is equivalent to some length c^{n2^m} RE E' for some constant c .

Proof.

1) find an n state, m variable BFA F equivalent to E (possible by Theorem 1a);

2) find an $n2^m$ state FA equivalent to F (possible by Theorem 2a);

3) find a c^{n2^m} length RE E' equivalent to F (possible by the standard method of converting FA into RE. See Ehrenfeucht and Zeiger (1976)).

Before proving Theorem 2d, we introduce some notation. The notions of normality, covering and index are from Ehrenfeucht and Zeiger (1976).

Definition. A labeled graph is a directed graph, with each arc labeled by a member of some given label set. There is at most one arc between two given nodes. A trail in a labeled graph is the sequence of labels along some connected path.

Definition. A RE E is normal w.r.t. graph A iff there are functions init and fin from subexpressions of E to nodes in A such that

1) If $F \cup G$ is a subexpression of E, *then* $\text{init}(F \cup G) = \text{init}(F) = \text{init}(G)$ and $\text{fin}(F \cup G) = \text{fin}(F) = \text{fin}(G)$.

2) If $F \cdot G$ is a subexpression of E then $\text{init}(F \cdot G) = \text{init}(F)$ and $\text{fin}(F \cdot G) = \text{fin}(G)$.

3) If F^* is a subexpression of E then $\text{init}(F^*) = \text{init}(F) = \text{fin}(F) = \text{fin}(F^*)$. $\text{init}(F^*)$ is called the base point of F^* .

4) If F is a subexpression of E then L(F) is contained in the set of trails from init(F) to fin(F) .

Definition. A RE E covers string s if s is a contiguous substring of some member of L(E) .

Definition. The index $I_s(E)$ of string s in RE E is the largest $k \geq 0$ such that E covers s^k, if such a k exists, and is ∞ when no such k exists. E is s-finite if $I_s(E) \neq 00$.

Definition. A labeled graph A is conservative if no loop path in A has the same trail as any other path in A . Hence, in a conservative graph, a loop path is uniquely determined by its trail.

Lemma 1. Let A be a conservative graph, s be a loop trail in A , and E be normal w.r.t. A . If E is s-finite, then $I_s(E) \leq \ell(E)$ (the length of E).

Proof. The proof is by induction on the length of E . The cases σ , $F \cup G$ and $F \cdot G$ are trivial, given

1) $I_s(\sigma) = 0$ or 1 for all s ,

2) $I_s(F \cup G) = \max(I_s(F), I_s(G))$,

3) $I_s(F \cdot G) \leq I_s(F) + I_s(G) + 1$,

all of which are obviously true. Suppose E is F^* . If the base point of F^* is not on the loop determined by s , then F^* cannot go around s any more times than F does. Suppose the base point of F^* is on the loop determined by s . If L(F) contains a cyclic permutation of s , then F^* is s-infinite, contrary to assumption. If on the other hand L(F) does not contain a cyclic permutation of s , then $I_s(F^*) \leq I_s(F) + 1$, which is enough to finish the induction.

Definition. A labeled graph A is forward deterministic (or just deterministic) if for every node u and label σ , there is at most one arc labeled σ leaving u . A is backward deterministic if the reversal of A , obtained by reversing all of the arcs in A , is deterministic.

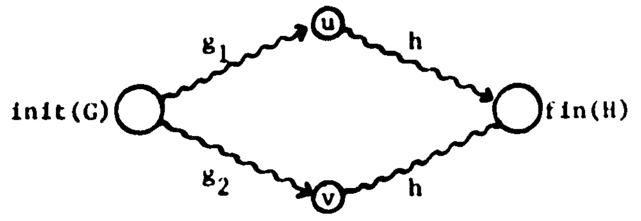
Lemma 2. Let A be forward and backward deterministic, and RE E be such that L(E) is contained in the set of trails from u to v in A , for some nodes u and v . Then E is normal w.r.t. A .

Proof. Let E have length n, and subexpression F of E have length k. We define $\mathrm{init}(F)$ and $\mathrm{fin}(F)$ inductively on $n - k$. For $k = n$ $(F = E)$, $\mathrm{init}(F) = u$ and $\mathrm{fin}(F) = v$.

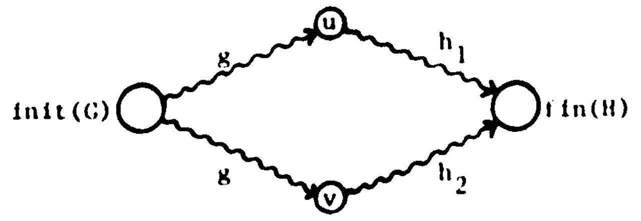
$G \cup H$. Suppose $\mathrm{init}(G \cup H)$ and $\mathrm{fin}(G \cup H)$ are already defined. Define $\mathrm{init}(G) = \mathrm{init}(H) = \mathrm{init}(G \cup H)$ and $\mathrm{fin}(G) = \mathrm{fin}(H) = \mathrm{fin}(G \cup H)$. Trivially, $L(G) \subseteq L(G \cup H) \subseteq$ trails from $\mathrm{init}(G)$ to $\mathrm{fin}(G)$, by induction. Similarly for H.

G^*. Suppose $\mathrm{init}(G^*) = \mathrm{fin}(G^*)$ has been defined. Define $\mathrm{init}(G) = \mathrm{init}(G^*) = \mathrm{fin}(G^*) = \mathrm{fin}(G)$. Trivially, $L(G) \subseteq L(G^*) \subseteq$ trails from $\mathrm{init}(G)$ to $\mathrm{fin}(G)$, by induction.

$G \cdot H$. Suppose $\mathrm{init}(G \cdot H)$ and $\mathrm{fin}(G \cdot H)$ have been defined so that $L(G \cdot H) \subseteq$ trails from $\mathrm{init}(G \cdot H)$ to $\mathrm{fin}(G \cdot H)$. Define $\mathrm{init}(G) = \mathrm{init}(G \cdot H)$ and $\mathrm{fin}(H) = \mathrm{fin}(G \cdot H)$. Suppose $L(G)$ contains two strings g_1 and g_2; and $L(H)$ contains h. A must contain the subgraph depicted below, since $g_1 \cdot h$ and



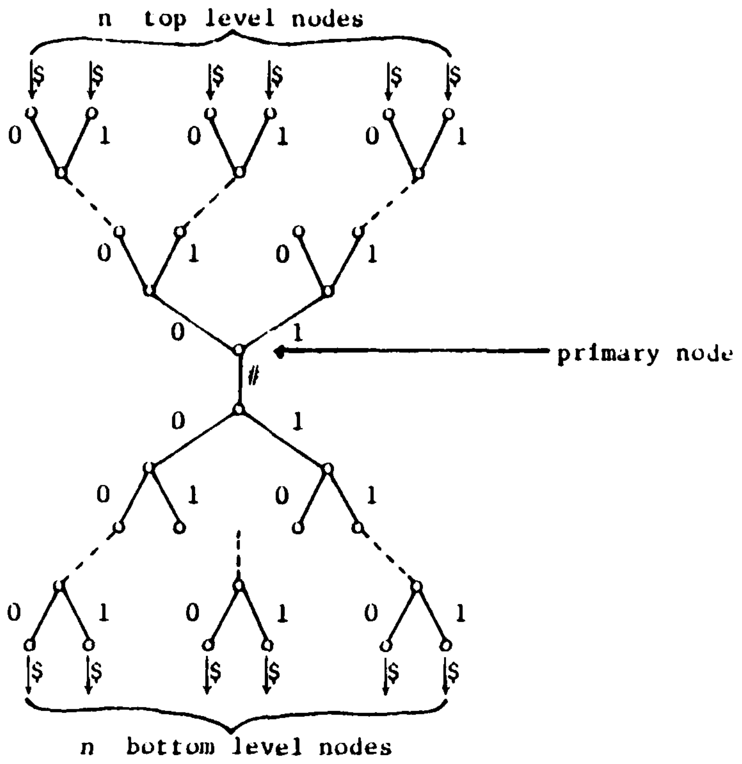
$g_2 \cdot h$ are both in $L(G \cdot H)$. By forward determinism, u and v are unique. By backwards determinism, $u = v$. Similarly, suppose $L(H)$ contains h_1 and h_2, and $L(G)$ contains g. Then A must contain the subgraph



By backward determinism, u and v are unique. By forward determinism, $u = v$. Hence, we see that all of the paths in $L(G)$ starting at $init(G)$ must end on $u = v$, which is just where all of the paths in $L(H)$ start. Define $fin(G) = u = v = init(H)$. Then $L(G) \subseteq$ trails from $init(G)$ to $fin(G)$, and $L(H) \subseteq$ trails from $init(H)$ to $fin(H)$.

Ehrenfeucht and Zeiger show that no short regular expression can represent all of the trails between two given nodes in the complete graph, where each arc bears a distinct label. We wish to modify the complete graph in such a way that it is still hard to describe for ordinary regular expressions, but is easy for Boolean regular expressions. We do that by effectively replacing the arc from u to v by a path whose trail is $\#\bar{v}\,\$\,\bar{u}^R$ where $\bar{u}$ and $\bar{v}$ are the binary representations of u and v, respectively. The nodes which correspond to nodes in the complete graph are called primary nodes; those which are on paths between primary nodes are called secondary nodes.

<u>Definition</u>. The <u>order n fan graph</u> FAN_n is defined for n a power of 2 as follows. There are n primary nodes, with $4n - 3$ secondary nodes associated with each primary node. A typical primary node and its harem of secondary nodes is pictured on the following page. All arcs point down. The primary nodes are numbered from 0 to $n - 1$. The $\$$ arcs are connected in such a way that there is a trail $\#\bar{v}\,\$\,\bar{u}^R$ from primary node u to primary node v, where $u, v \in \{0, \ldots, n - 1\}$, and $\bar{u}$ $(\bar{v})$ is the $\log n$ digit binary representation of u (v).



Definition. An m-restriction of FAN_n , for $m \leq n$, is a subgraph of FAN_n containing m primary nodes, such that the length $2\log n + 2$ paths between existing primary nodes are intact, and all extraneous nodes are absent.

If A is an m-restriction of FAN_n , then A is clearly forward and backward deterministic. A is also conservative, for every loop passes through a primary node, and immediately after leaving a primary node, a trail names the node just left. The # symbol indicates the start of a node name.

Theorem 3. For every $n = 2^k$, every m-restriction A of FAN_n , $m \leq n$, and every primary node u in A , there is a trail p in A from u to itself such that if E is any regular expression covering p , and which is normal w.r.t. FAN_n , then $\ell(E) \geq 2^m$.

Proof. By induction on m. In what follows, "normal" means normal w.r.t. FAN_n.

$m = 1$. A 1-restriction of FAN_n is just a loop containing $2\log n + 2$ nodes. That loop has length at least $2^m = 2$.

$m > 1$. Let $u_0, \ldots, u_{m-1}$ be the primary nodes in A. Let A_i, $i = 0, \ldots, m-1$ be the $(m-1)$-restriction of FAN_n obtained by deleting u_i and associated secondary nodes from A. Let $\ominus$ and $\oplus$ be substraction and addition mod m. By induction, every A_i contains a trail p_i from $u_{i\ominus 1}$ to itself such that every normal E covering p_i has length at least 2^{m-1}. For $j = 0, \ldots, m-1$, define

$$q_j = (p_{j\oplus 1})^k \, a_{j,j\oplus 1} \, (p_{j\oplus 2})^k \, a_{j\oplus 1,j\oplus 2} \cdots (p_{j\oplus m})^k \, a_{j\oplus m-1,j} \, ,$$

where $k = 2^m$ and a_{ij} is a length $2\log n + 2$ trail from node i to node j. q_j is a loop from node j to itself in A.

To prove Theorem 3 we need to show that, for $j = 0, \ldots, m-1$, if E is normal and covers q_j, then $\ell(E) \geq 2^m$. Suppose E is normal and covers q_j. Then for every i, $l_{p_i}(E) \geq 2^m$. If E is p_i-finite for some i, then by Lemma 1 $\ell(E) \geq 2^m$, which is what we want to show.

Suppose on the other hand that E is p_i-infinite for all i. In what follows, "minimal" means "minimal with respect to the relation 'subexpression of'". Since E has finitely many subexpressions, for each i there is a minimal subexpression of E which is p_i-infinite. Such a subexpression must be a star, say F_i^*. Choose a minimal F_r^* from among the F_i^*.

F_r^* is normal and covers p_r in A_r, so by induction $\ell(F_r^*) \geq 2^{m-1}$.

The subexpression F_r^* of E has a base point v in FAN_n by the normality of E. The path for which p_r is a trail must pass through the base point of F_r^*, for otherwise either F_r^* would not be p_r-infinite, or F_r would be p_r infinite, violating the minimality of F_r^*. Hence, v is on the path determined by p_r, that is, v is in A.

Let E/F_r^* be obtained from E by replacing F_r^* by the expression λ representing the null string. By definition, $\ell(\lambda) = 0$. By supposition E is p_v-infinite. But the path determined by p_v does not pass through node v, while every member of F_r^* determines a path which does pass through v. Hence, E/F_r^* must still be p_v-infinite. Thus E/F_r^* covers p_v in A_v, and by induction $\ell(E/F_r^*) \geq 2^{m-1}$. Putting that fact together with $\ell(F_r^*) \geq 2^{m-1}$ gives the desired result, $\ell(E) \geq 2^m$.

Theorem 2d. For infinitely many m there is a BRE E of length m which is not equivalent to any RE E' of length less than $2^{2^{dm}}$ for some $d > 0$.

Proof. Let E be a BRE for the set of all trails from primary node 0 in FAN_n to itself. E can use variables $x_1, \ldots, x_k$ and $y_1, \ldots, y_k$, $k = \log n$, to represent integers x and y respectively, in the range $[0, \ldots, n-1]$. Let "x←random" be the expression $(x_1\uparrow \cup x_1\downarrow)\cdot(\cdots)\cdot(x_k\uparrow \cup x_k\downarrow)$. Let "generate x" be the expression $(\bar{x}_1?\cdot 0 \cup x_1?\cdot 1)\cdot(\cdots)\ (\bar{x}_k?\cdot 0 \cup x_k?\cdot 1)$, and "generate y^R" be $(\bar{y}_k?\cdot 0 \cup \bar{y}_k?\cdot 1)\cdot(\cdots)\cdot(\bar{y}_1?\cdot 0 \cup y_1?\cdot 1)$.

E is defined by

$$E = (y \leftarrow 0) \cdot ((x \leftarrow \text{random}) \cdot \# \cdot (\text{generate } x) \cdot \$ \cdot (\text{generate } y^R) \cdot (y \leftarrow x))^* \, (y = 0?) \ .$$

E has length $c \log n$ for some c . Suppose E' is equivalent to E . By Lemma 2 E' is normal w.r.t. FAN_n . By Theorem 3, $\ell(E') \geq 2^n$. Let $m = c \log n$. Then $\ell(E') \geq 2^{2^{dm}}$ for $d = 1/c$.

This concludes the proof of Theorem 2d.

The result of Ehrenfeucht and Zeiger, that the set of paths in some n node graphs can only be described by length 2^{n-1} regular expressions, requires alphabets of unbounded size. As the proof given here uses a 4-symbol alphabet, we can give a bound which applies when the alphabet size is bounded. However, because FAN_n has $O(n^2)$ nodes, the result for bounded alphabets is weaker than that for unbounded alphabets.

<u>Theorem 4</u>. For infinitely many m there is an m state FA F over a 4-element alphabet which is equivalent to no RE E of length less than $d^{\sqrt{m}}$ for some $d > 1$.

<u>Proof</u>. Let F be the automaton induced by FAN_n , where node 0 is both the start and final state. F has cn^2 states for some c . Any RE E for the set of trails from node 0 to itself must have length at least 2^n . Let $m = cn^2$. Then $L(E) \geq d^{\sqrt{m}}$ for $d = 2^{\sqrt{1/c}}$.

4. OPEN QUESTIONS

The lower bounds 2b and 2d require length n expressions with $O(n)$ Boolean variables. Is it possible to make the number of Boolean variables a parameter, which can vary independently of n , as in Theorems 2a and 2c ? Our upper bound on Boolean variables removal from regular expressions is very poor when $m = 0$, for our method causes an exponential blowup when no work at all is required. Suppose m is not 0, but is small, for instance $m = 1$. Then is the exponential blowup really required? If so, then a single Boolean variable is extremely powerful. If not, then where does the double exponential behavior of Theorem 2d take hold?

The $d^{\sqrt{m}}$ lower bound of Theorem 4 is weaker than the 2^{m-1} bound proved by Ehrenfeucht and Zeiger for unbounded alphabets. Can a d^m bound be proved for bounded alphabets?

References

1. K.R. Abrahamson, "Decidability and expressiveness of logics of processes," Ph.D. Thesis, University of Washington, Seattle, Washington, 1980.

2. V.A. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Massachusetts.

3. R.S. Cohen, "Star height of certain families of regular events," JCSS 4 (1970), 281-297.

4. A. Ehrenfeucht and P. Zeiger, "Complexity measures for regular expressions," JCSS 12 (1976), 134-146.

5. L.J. Stockmeyer, "The complexity of decision problems in automata theory and logic," Ph.D. Thesis, Massachusetts Institute of Technology, 1974.

DISTRIBUTION LIST

Office of Naval Research Contract N00014-80-C-0221
Michael J. Fischer, Principal Investigator

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
(12 copies)

Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Dr. R. B. Grafton, Scientific Officer (1 copy)
Information Systems Program (437) (2 copies)
Code 200 (1 copy)
Code 455 (1 copy)
Code 458 (1 copy)

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106
(1 copy)

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375
(6 copies)

Office of Naval Research
Resident Representative
University of Washington, JD-27
422 University District Building
1107 NE 45th Street
(1 copy)

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
Code RD-1
Washington, D.C. 20380
(1 copy)

Naval Ocean Systems Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152
(1 copy)

Mr. E. H. Gleissner
Naval Ship Research and Development Center
Computation and Mathematics Dept.
Bethesda, MD 20084
(1 copy)

Captain Grace M. Hooper (008)
Naval Data Automation Command
Washington Navy Yard
Building 166
Washington, D.C. 20374
(1 copy)

Defense Advanced Research Projects Agency
Attn: Program Management/MIS
1400 Wilson Boulevard
Arlington, VA 22209
(3 copies)